

AD-A053 245

YALE UNIV NEW HAVEN CONN DEPT OF COMPUTER SCIENCE  
EVALUATION CRITERIA FOR PROCESS SYNCHRONIZATION.(U)  
JUN 75 R J LIPTON, L SNYDER, Y ZALCSTEIN  
RR-90

F/G 9/2

N00014-75-C-0752

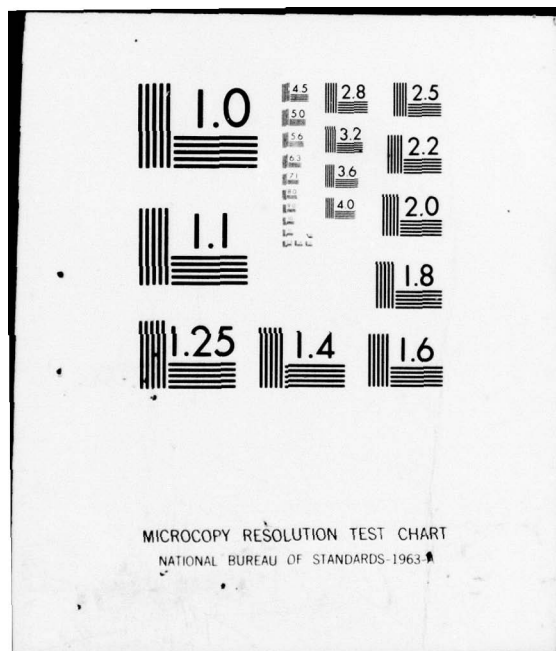
NL

UNCLASSIFIED

| OF |  
AD  
A053245



END  
DATE  
FILMED  
6-78  
DOC



AD NO. \_\_\_\_\_  
DDC FILE COPY

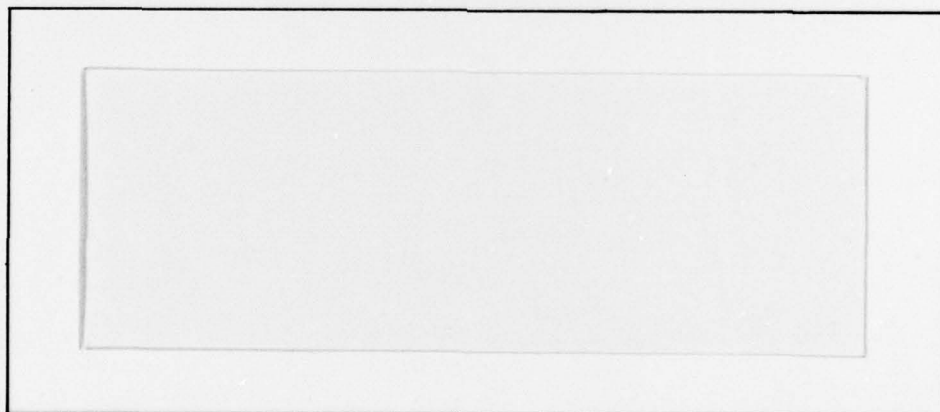
AD A 053245



12  
B.S.

*See  
Horn  
1/173*

**DISTRIBUTION STATEMENT A**  
Approved for public release;  
Distribution Unlimited



YALE UNIVERSITY  
DEPARTMENT OF COMPUTER SCIENCE

DDC FORM 1	
TYPE	DDC Section <input checked="" type="checkbox"/>
DATE	DDC Section <input type="checkbox"/>
UNCLASSIFIED	<input type="checkbox"/>
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist. Avail. and SPECIAL	
A	<i>[scribble]</i>

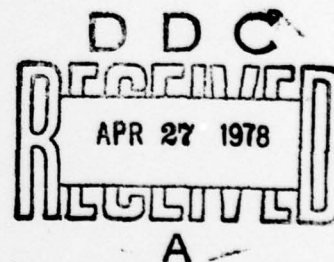


**DISTRIBUTION STATEMENT A**  
 Approved for public release;  
 Distribution Unlimited

# Evaluation Criteria for Process Synchronization

R. J. Lipton,<sup>†</sup> L. Snyder<sup>††</sup>  
 and Y. Zalcstein<sup>†††</sup>

Research Report #90



<sup>†</sup> Department of Computer Science, Yale University, New Haven, Connecticut 06520. Part of this work was done while at IBM Research Center at Yorktown Heights, and part was supported by ONR under grant N00014-75-C-0752.

<sup>††</sup> Department of Computer Science, Yale University, New Haven, Connecticut 06520. Supported, in part, by ONR under grant N00014-75-C-0752.

<sup>†††</sup> Supported, in part, by NSF grant DCR75-01998.

# 1975 SAGAMORE COMPUTER CONFERENCE ON PARALLEL PROCESSING

## EVALUATION CRITERIA FOR PROCESS SYNCHRONIZATION

R. J. Lipton<sup>+</sup>

L. Snyder<sup>++</sup>

Computer Science Department  
Yale University  
New Haven, Connecticut 06520

Y. Zalcstein<sup>+++</sup>

Computer Science Department  
State University of New York  
Stony Brook, New York 11794

**Abstract** -- While there are by now well-established criteria for evaluating serial algorithms, such as space and time measures, these criteria cannot be readily applied to asynchronous algorithms. We propose a method for the evaluation of the performance of an asynchronous algorithm. This method is based on the study of delays that are often introduced when one solves a synchronization problem. We then illustrate this method by proving results about the efficiency of various solutions to synchronization problems.

### 1. Introduction

A central problem in computer science is that of evaluating competing algorithms for the same task. In the case that the algorithms are to be executed sequentially, several evaluation criteria are commonly used. First, it is easy to express the idea that two algorithms "do the same thing" by the requirement that they have the same input-output behavior. Secondly, given that two algorithms have the same input-output behavior, they may be compared by considering the execution time required, memory space required, numerical (or other type of) stability and so on. By contrast asynchronous algorithms cannot be evaluated so easily, due to several important reasons.

First, asynchronous algorithms -- especially those used in operating systems -- are not necessarily supposed to halt. Indeed, considerable effort is often required to guarantee that they do not halt, i.e. do not deadlock or crash. Therefore, it often makes no sense to discuss the input-output behavior of these asynchronous algorithms. Thus it is not at all clear when two such algorithms "do the same thing".

Another difficulty is that of measuring efficiency. Simply counting the number of steps

required to accomplish a task does not reflect the utilization of multiple processors. Are algorithms requiring more steps -- which can be done in parallel -- to be preferred over those requiring fewer steps -- which cannot be done in parallel? Similarly, algorithms requiring less memory are not clearly superior if referencing this memory causes processor interference.

Since we are mainly concerned with synchronization, the questions of efficiency can be stated as: How much overhead is required (and how much is acceptable) to accomplish process synchronization? Will the method we chose to solve our synchronization problem cause delays or interference which are unacceptable?

In this paper we present a criterion for evaluating asynchronous algorithms. Rather than attempt to assign absolute measures of resource utilization -- a task that may well be impossible to do in a useful way -- we define, relative to a suitable measure of time, for each non-negative integer  $k$ , a relation,  $\text{simulate}_k$ , between asynchronous algorithms. For asynchronous algorithms  $Q$  and  $P$

$Q \text{ simulate}_k P$

will mean that there is a mapping from computations (state changes) in  $Q$  to computations in  $P$ . This consideration of state changes avoids the difficulty of non-halting algorithms not being input-output comparable. The efficiency of this correspondence (i.e. the amount of overhead  $Q$  requires to accomplish the same effect as  $P$ ) is measured by the integer  $k$ .  $k$  measures how closely the "parallelism" of  $Q$  and  $P$  are related. When  $k = 0$ ,  $Q$  uses multiprocessors as efficiently as  $P$ , but as  $k \rightarrow \infty$ ,  $Q$  uses multiprocessors less and less efficiently. Thus there will be a sequence of relations

$\text{simulate}_0, \text{simulate}_1, \dots$

which allow increasing freedom with a corresponding decrease in efficiency.

### 2. The Model

We have used a more "program oriented" model to study related problems ([5] - [8]). However, experience has shown that, as far as the analysis of synchronization is concerned, it is possible to abstract the model further to the language theoretic one which we present below.

(+) Part of this work was done while at IBM Research Center at Yorktown Heights, and part was supported by ONR under Grant N00014-75-C-0752.

(++) Supported, in part, by ONR under Grant N00014-75-C-0752.

(+++)

Supported, in part, by NSF Grant DCR75-01998.

The model will ignore such issues as what kind of language the algorithm is specified in, how the actual scheduling is determined, and, most importantly, how the algorithms are actually implemented. These are, of course, important considerations, but it is our contention that a study of the logical implementation of asynchronous programs is of prime importance.

Let  $\Sigma$  be a finite set. Elements of  $\Sigma$  will be thought of as actions (instructions or statements). Informally, a computation is any sequence of actions that respects the control flow of an asynchronous program  $P$  (we assume fixed initial values of all variables so that different sequences represent true asynchronous behavior and are not merely a reflection of different inputs to  $P$ ). Clearly, if  $x$  is a computation, then so is any prefix (initial subsequence) of  $x$ . We formalize this notion as follows.

**Definition:** Let  $\Sigma$  be a finite non-empty set. An asynchronous program is a subset  $P$  of  $\Sigma^*$ , the set of all sequences of elements of  $\Sigma$ , which is closed under the operation of taking prefixes. Elements of  $\Sigma$  are called actions and elements of  $P$  are called computations.

**Definition:** Let  $P \subseteq \Sigma_P^*$  be an asynchronous program. A cost function is a function  $c: \Sigma_P^* \rightarrow N$ , where  $N$  is the set of non-negative integers which is additive with respect to concatenation, i.e.  $c(xy) = c(x) + c(y)$ . Intuitively,  $c$  measures "time".

Let  $P \subseteq \Sigma_P^*$  be an asynchronous program and  $c$  be a cost function. Define a delay function.

$$d_c: \Sigma_P^* \times \Sigma_P^* \rightarrow N \cup \{\infty\} \text{ by}$$

$$d_c(x, f) = \min \{c(y) : y \in \Sigma_P^* \text{ and } xy \in P\}, \text{ where}$$

$$d_c(x, f) = \infty \text{ if there is no such } y. \text{ If}$$

$$c(x) = \text{length}(x) \text{ we will denote } d_c \text{ by } d.$$

$d_c(x, f)$  measures the minimal amount of "time" as measured by  $c$  that must elapse before  $f$  can execute following  $x$ . This quantity is important for several reasons. In a real time system, the value of  $d_c(x, f)$  may be critical to the correctness of the system. Also, given additional structure in the model, the delay function acts as a quantitative measure of how well multiprocessors can be utilized.

When comparing two asynchronous programs  $P \subseteq \Sigma_P^*$  and  $Q \subseteq \Sigma_Q^*$ , it is convenient to think of one of them, say  $Q$ , as implementing the effect of  $P$  by using more primitive operations. According to this view,  $Q$  is the "compiled" or "macro

expanded" version of  $P$ . One can then consider a mapping  $M$  from  $P$  to  $Q$  representing this compilation process. In the model presented here, it will be more convenient to consider the "inverse", say  $h$ , of  $M$ , from  $Q$  to  $P$ . Thus a sequence of actions  $\alpha\beta$  in  $Q$  will be the "expansion" of a single action  $g$  in  $P$ . The action  $f$  will be considered to implement the action  $g$  while  $\alpha$  and  $\beta$  will be considered as bookkeeping operations or overhead.

Our model also requires that  $h$  be a homomorphism which simply means that flow of control in  $Q$  is a copy of the flow of control of  $P$ .

Formalizing the discussion above, we obtain the following

**Definition:** Let  $Q$  and  $P$  be asynchronous programs, i.e.  $Q \subseteq \Sigma_Q^*$  and  $P \subseteq \Sigma_P^*$ . Then  $h$  is a decoder from  $Q$  to  $P$  provided  $h$  is a string morphism from  $\Sigma_Q^*$  into  $\Sigma_P^*$ , i.e.,  $h(xy) = h(x)h(y)$  for all  $x, y$  in  $\Sigma_Q^*$  and  $h(f) \in \Sigma_P \cup \{\Lambda\}$  for all  $f \in \Sigma_Q$  where  $\Lambda$  is the empty string and  $h(Q) = P$ .  $f \in \Sigma_Q$  is called observable if  $h(f) \neq \Lambda$ , otherwise it is called a bookkeeping action.

We can now define simulate<sub>k</sub>.

**Definition:** Let  $Q$  and  $P$  be asynchronous programs over alphabets  $\Sigma_Q$  and  $\Sigma_P$  respectively, and let  $c$  be a cost function on  $\Sigma_Q$ . Then

$$Q \text{ simulate}_k P$$

provided that there is a decoder  $h$  from  $Q$  to  $P$  such that for all  $x \in \Sigma_Q^*$  and  $f \in \Sigma_Q$ ,  $f$  observable,

$$h(x)h(f) \in P \text{ implies } d_c(x, f) \leq k.$$

Intuitively, if, after a sequence of actions  $h(x)$ ,  $P$  is not "stopped" i.e. some action  $g$  may proceed, then  $Q$  may be stopped at  $x$  but only temporarily, in the sense that there is a bound  $k$  on the amount of time, as measured by  $c$ , that must elapse before the action  $f$  corresponding to  $g$  can be "released". This is our measure of efficiency.

The smallest  $k$  such that  $Q$  simulate<sub>k</sub>  $P$  will be denoted by delay ( $Q, P$ ).

### 3. Examples and discussion

In this section we illustrate the preceding definitions by examples from the literature. To simplify the discussion, we will use a suggestive informal notation, as is commonly employed in the synchronization literature. It should be pointed out that the advantage of the abstract definition of an asynchronous program is its conceptual economy and aid in simplifying proofs. For describing particular examples, a "program oriented" notation is clearly preferable. This is

quite analogous to the description of languages by grammars.

Consider the following asynchronous programs which we take as defining the semantics of the "first reader-writer problem" of [1] (for a discussion of the semantics of synchronization problems see [5], [6]).

```

reader-i (1 ≤ i ≤ n)      writer
ci: P(S)                  j: P(S|n)
ei: read                  k: write
hi: V(S)                  l: V(S|n)

```

where  $S$  is a global variable (semaphore) whose initial value is  $n$  and  $P, V$  are Dijkstra's primitives, while  $P(S|n)$ ,  $V(S|n)$  are the generalizations of these primitives [9].  $P(S|n)$  is an indivisible action of the form

when  $S \geq n$  do  $S \leftarrow S - n$

the assignment  $S \leftarrow S - n$  is executed only when  $S \geq n$ , otherwise, control is interrupted until such time as  $S \geq n$  is satisfied.  $V(S|n)$  is an indivisible action of the form

when true do  $S \leftarrow S + n$ .

Each of the processes reader i and writer is cyclic so that for example,  $j$  can proceed after execution of  $kl$ . Let us denote the set of computations of this program by  $P$ . For example,  $c_1c_2 \in P$ , while  $jc_2 \notin P$ .

Now let  $Q_1$  be the asynchronous program of figure 1. This program corresponds to the solution to the first reader-writer problem found in [1].

integer readcount; (initial value 0)

semaphore M, W; (initial value 1)

reader-i 1 ≤ i ≤ n

A<sub>i</sub> : P(M)

B<sub>i</sub> : readcount ← readcount + 1

C<sub>i</sub> : if readcount = 1 then P(W)

D<sub>i</sub> : V(M)

E<sub>i</sub> : read

F<sub>i</sub> : P(M)

G<sub>i</sub> : readcount ← readcount - 1

H<sub>i</sub> : if readcount = 0 then V(W)

I<sub>i</sub> : V(M)

writer

J : P(W)

K : write

L : V(W)

Figure 1. First solution to reader-writer problem.

We will now study the relationship between  $Q_1$  and  $P$ . First let  $h$  be the mapping defined by:

$h(C_i) = c_i$

$h(E_i) = e_i$

$h(H_i) = h_i$

$h(J) = j$

$h(K) = k$

$h(L) = l$

$h(X) = \Lambda$  for all other actions  $X$ .

It is not difficult to verify that  $h(Q) = P$ . For instance the computation

$A_1B_1C_1D_1A_2B_2C_2D_2$

maps under  $h$  to  $c_1c_2$ .

We wish to measure the efficiency of this solution. First, we claim that for the given decoder  $h$ ,  $Q_1$  simulate<sub>k</sub>  $P$  implies  $k \geq 3$ . To see this, take  $x = A_1B_1C_1$  and  $f = C_2$ , then the shortest  $y$  such that  $xyfeQ$  is  $D_1A_2B_2$ , which exits from the critical section of reader-1 restores the semaphore  $M$  to 1 and then enters the critical section  $A_2-D_2$  of reader-2. Thus  $d(x, f) = 3$ , while  $h(x)h(f) = c_1c_2 \in P$ . By a straightforward analysis of cases, based on the observation that one need execute at most three actions between two "successive" observables, it follows that, for this decoder,  $k \leq 3$ . Next, we claim that under no decoder  $h_1$ , can either  $A_i$  or  $B_i$  be observable actions. Assume the contrary and let  $h_1(A_i) = c_i$ . Since  $A_1J \in Q$  and since clearly  $h_1(J) = j$ ,  $h(A_1J) = h(A_1)h(J) = c_1j \notin P$ , which is a contradiction since  $h$  maps computations into computations. Similarly,  $h_1(B_i) \neq c_i$ . Thus either  $h_1(C_i) = c_i$  for all  $i$  and we can argue as before that  $k \geq 3$ , or, for some  $i$ ,  $h_1(D_i) = c_i$ , but, in the latter case  $d(A, D_i) = |A_iB_iC_i| = 3$  so  $k \geq 3$  in all cases. Hence delay  $(Q_1, P) = 3$ .

Thus  $Q_1$  introduces new delays, but delay  $(Q_1, P)$  is fixed, independently of the number  $n$  of readers.

Let us now compare  $Q_1$  with an alternative solution,  $Q_2$ , represented in figure 2.

$$S_1 = S_2 \dots = S_n = 1 \text{ initially}$$

reader  $i$      $1 \leq i \leq n$

$$C_i: P(S_i)$$

$E_i$ : read

$$H_i: \quad V(S_i)$$

writer

$$J_1: \quad P(S_1)$$

• •

$$J_n: P(S_n)$$

K: write

$$L_1: \quad v(S_1)$$

•

$$L_n: \quad v(s_n)$$

Figure 2. Second solution to reader-writer problem.

Clearly any decoder  $h$  from  $Q_2$  to  $P$  will have to map  $h(C_1) = c_i$ ,  $h(E_1) = e_i$ ,  $h(H_1) = h_i$  and  $h(K) = k$ . If  $h(J_1) = j$ ,  $i \neq 1$ , then for  $x = J_1$  and  $f = C_1$ ,  $d(x, f) = n + 1$  and  $h(xf) = c_i \in P$ , while if  $h(J_1) = j$ , for  $x = J_2$  and  $f = C_2$ ,  $d(x, f) = n + 1$ . Thus  $\text{delay}(Q_2, P) \geq n + 1$ .

Thus  $Q_2$  introduces delays that are unbounded as a function of the number of readers present. Therefore, delay  $(Q,P)$  is a quantitative measure of efficiency which agrees with the intuition that  $Q_1$  is a better solution than  $Q_2$ .

The above differences become even more interesting if we allow different cost functions.

For example, we may want to use a weighted length function  $c$ . Observing that most of the time is actually spent in the "read" and "write" sections of the program, we may assign to the "read" and "write" actions weight  $t > 1$  while all other actions in  $Q_i$  and  $P$  get assigned weight 1.

With respect to this cost function, delay  $(Q_1, P)$  is still 3. However, delay  $(Q_2, P) \geq n + t$  since for the above values of  $x$  and  $f$ , the program has to go through the "write" section in order to release the reader.

#### 4. Existence Theorems

In this section we give proofs of various simulation results concerning Dijkstra's P and V primitives.

In our previous work [5], [6], [8], we have shown that with respect to a suitable notion of

"simulate", FV systems are too weak and cannot simulate even rather simple synchronization problems. Many readers of our work objected to "simulate" as being too strong, based on the intuitive feeling that FV is "universal". Using the "simulate<sub>k</sub>" relation, we now show that FV is "universal" in the sense that for any asynchronous program P, there is a FV program Q such that Q simulate<sub>k</sub> P, for some k. However, k grows unboundedly as a function of the size of P.

In the following, we will use the when...do notation introduced in [5]:

when  $\beta$  do  $\theta$

where  $\beta$  is a predicate and  $\Theta$  is a statement means that  $\Theta$  is executed only if  $\beta$  is true. Otherwise, control is interrupted until such time as  $\beta$  is true.

Definition: A PV asynchronous program is an asynchronous program  $P$  such that there is a distinguished subset  $\mathcal{A}$  of the program variables (the elements of  $\mathcal{A}$  are called semaphores) which can only be used by actions of the form  $P(S)$  or  $V(S)$ ,  $S \in \mathcal{A}$ ,  $\{2\}$  where

P(S) is    when  $S > 0$  do  $S \leftarrow S - 1$     and

```
V(S) is    when true do S ← S + 1
```

Theorem 1. For every asynchronous program P, there exists a non-negative integer k and a PV asynchronous program Q such that Q simulates<sub>k</sub> P, with length as cost function.

Proof. Let  $P$  be an asynchronous program and suppose  $P$  consists of  $n$  actions of the form

(1) when  $\beta_1$  do  $\theta_1$

(n) when  $\beta_n$  do  $\theta_n$

We construct Q as an asynchronous program containing  $n + 1$  processes, where the first  $n$  are constructed from the  $n$  actions of P and the  $n + 1$ -st is a "monitor". The monitor can actually be incorporated into the individual processes, but its isolation as a separate processes enhances the efficiency and readability of the program.

For the  $i$ -th action of  $P$ , construct the following process in  $Q$ :

```

process-i
(1)  Li:      P(Si)
(2)                P(S)
(3)                w + w + 1
(4)                if w = np then V(E)
(5)                V(S)
(6)                P(E)
(7)                if ok = 1 then
(8)                θi
(9)                ok + 0
(10)               w + w - 1
(11)               if w > 0 then V(E)
(12)               else V(M)
(13)               goto Li;

```

Where S is a mutual exclusion semaphore (initial value 1) used to protect the critical section (2)-(5), E is a semaphore (initial value 0) used to release all actions that may execute at a given step (the "ready-set" in the terminology of [5]). M is a semaphore (initial value 1) used for communication with the monitor. S<sub>i</sub> is a local semaphore (initial value 0) which is enabled at a given step if the i-th action may execute at that step. The variable w is a counter, np is a variable giving the number of actions that may execute at any step and ok is a flag.

The monitor process is given by

```

LM:      P(M)
          ok + 1
          np + φ(β1, ..., βn)
          t + ψ(β1, ..., βn)
          if q(t,1) then V(S1)
          :
          if q(t,n) then V(Sn)
          goto LM;

```

φ is a function that computes the number of processes that may execute given the values of the β<sub>i</sub>'s and ψ is a function that figures out which processes may execute and encodes this information into t. q(t,i) will decode t and enable S<sub>i</sub> accordingly.

The monitor starts and enables some process-i then enters its critical section (2)-(5). Inside

the critical section it acknowledges that it is ready to execute and waits on (6) for release. If all pending processes have so acknowledged, one of them is enabled in line (4). Note that the scheduling responsibility has not been usurped by this simulation in the sense of deciding which process will execute next.

Assuming process j is the first to execute beyond line (6), and since ok will be 1, it will execute θ<sub>j</sub>, disable all others from executing (9) (since executing θ<sub>i</sub> could have changed which processes may now proceed), acknowledges that it has passed (10) and then releases another process if not all have been released (11), otherwise it releases the monitor (12).

Let h((8)<sub>i</sub>) = i and h(r) = A for all other actions in Q. Evidently h(Q) = P. To bound the efficiency of the simulation, observe that between any two consecutive observable actions (8)<sub>i</sub> and (8)<sub>j</sub>, r bookkeeping actions are executed, where

$$r \leq 5u + 2n + 8 + 5v$$

where u is the number of processes that may execute after (8)<sub>i</sub> and v the number that may execute after (8)<sub>j</sub>. Since u, v ≤ n, k is bounded by 12n + 8.

The proof of Theorem 1 suggests another cost function -- the number of observable actions in a word. Let us denote this cost function by c<sub>1</sub>. Then we have the following:

**Corollary.** For every asynchronous program P, there exists a PV asynchronous program Q such that Q simulate<sub>0</sub> P with cost function c<sub>1</sub>.

**Proof.** Immediate from the proof of Theorem 1, since there are only bookkeeping actions between (8)<sub>i</sub> and (8)<sub>j</sub>.

In [8], we have shown that FV systems with only binary ({0,1}-valued) semaphores are strictly weaker than PV systems in the sense that there are FV systems that cannot be simulated by any FV system with only binary semaphores. However, we have the following

**Theorem 2.** For every FV asynchronous program P, there is a PV asynchronous program Q with only binary semaphores such that Q simulate<sub>3</sub> P, with length as cost function.

# 1975 SAGAMORE COMPUTER CONFERENCE ON PARALLEL PROCESSING

Proof. The construction is essentially sketched in [2]. For each semaphore  $S$  in  $P$ , add a new mutual exclusion semaphore  $E$  (initial value 1) and a new integer variable  $x$  (initial value 0).  $Q$  is obtained from  $P$  by replacing each  $P(S)$  by

```
P(E)
x ← x - 1
if x < 0 then begin V(E);
P(S) end
else V(E);
```

and  $V(S)$  by

```
P(E)
x ← x + 1
if x ≤ 0 then V(S)
V(E)
```

Let the third line in each expansion be the observable action. Then clearly  $h(x)h(f) \in P$  implies  $d(x, f) \leq 4$ .

## Acknowledgements

We would like to thank M. Condry, E. Ekanadham, P. Henderson, N. Pippenger, and A. Silberschatz for several helpful discussions.

## References

- [1] P.J. Courtois, F. Heymans and D.L. Parnas, "Concurrent control with 'readers' and 'writers'", CACM 14 (Oct. 1971) pp. 667-668.

- [2] E.W. Dijkstra, "Cooperating sequential processes", in Programming Languages (F. Genuys, ed.), Academic Press, pp. 43-112.
- [3] R.M. Karp and R.E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing", SIAM J. Applied Math. 14 (1966) pp. 1390-1411.
- [4] R.M. Karp and R.E. Miller, "Parallel program schemata", J. Comput. Syst. Sci. 3 (1969), pp. 147-195.
- [5] R.J. Lipton, "Limitations of synchronization primitives with conditional branching and global variables", Proc. Sixth ACM Symposium on Theory of Computing (1974) pp. 230-241.
- [6] R.J. Lipton, Limitations of synchronization primitives, Research Report #31, Computer Science Department, Yale University, (Aug. 1974), 51 pp.
- [7] R.J. Lipton, "Reduction: A new method for proving properties of systems of processes", Conference Record of the Second Annual ACM Symposium on Principles of Programming Languages (1975), pp. 78-86.
- [8] R.J. Lipton, L. Snyder and Y. Zalestein, "A comparative study of models of parallel computation", Conference Record of the Fifteenth Annual IEEE Symposium on Switching and Automata Theory (1974), pp. 145-155.
- [9] H. Vantilborgh and A. van Lamweerde, "On an extension of Dijkstra's semaphore primitives", Information Processing Letters 1 (1972) pp. 181-186.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) BR-98	2. GOVT ACCESSION NO. (9)	3. RECIPIENT'S CATALOG NUMBER Research rept.
4. TITLE (and Subtitle) (6) Evaluation Criteria for Process Synchronization		5. TYPE OF REPORT & PERIOD COVERED Technical
7. AUTHOR(s) (10) Richard J. Lipton, Lawrence Snyder, Y. Zalcstein		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University Department of Computer Science 10 Hillhouse Ave, New Haven, CT 06520		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0752 NSF-DCR75-01998
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, Virginia 22217		10. PROGRAM ELEMENT PROJECT TASK AREA & WORK UNIT NUMBERS (11)
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 9p.		12. REPORT DATE JUN 1975
		13. NUMBER OF PAGES
		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Distribution of this report is unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) synchronization asynchronous computation operating systems cooperating processes simulate synchronization primitives		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) While there are by now well-established criteria for evaluating serial algorithms, such as space and time measures, these criteria cannot be readily applied to asynchronous algorithms. <del>We propose</del> <sup>is proposed</sup> a method for the evaluation of the performance of an asynchronous algorithm. This method is based on the study of delays that are often introduced when one solves a synchronization problem. <del>We then illustrate</del> <sup>is illustrated</sup> this method by proving results about the efficiency of various solutions to synchronization problems.		

407054